Debugging Programs 12

This chapter describes the THINK Project Manager's source-level debugger. The source-level debugger lets you debug your application the way you wrote it: in source code. The debugger lets you step through your program line by line, set breakpoints, and examine and set the values of your variables.

This chapter describes some of the more advanced features of the THINK Debugger. "Tutorial: Bullseye," Chapter 5, teaches you how to use the debugger. You might want to work through that tutorial before you read this chapter.

Before you begin

Make sure the THINK Debugger is in the same folder as the THINK Project Manager. To use the source debugger, you need at least four megabytes (4MB) of memory, and you must be running the THINK Project Manager under System 7 or MultiFinder. To debug large projects, you need more memory. The debugger works only with application projects. It won't work with code resources or device drivers.

Contents

Running with the Debugger 217
- Turning the debugger on 217
- Running the project 218

The Debugger Windows. 219
- The Source window. 219
- The Data window 220

Working with the Source Window 221
- The current statement and the selected statement 222
- The current function indicator 223
- Viewing other source files in the Source window 224
- Viewing header files in the Source window 224
- Editing files while debugging 226
- Searching in the Source window 227

Setting Breakpoints 228
Simple breakpoints 229
Setting breakpoints in another file 230
Setting temporary breakpoints 230
Setting conditional breakpoints 231
Using Debugger() and DebugStr() 233

Controlling Execution 233
Go 234
Trace 234
Step 234
Step In. 234
Step Out 234
Stop 235
Go Until Here 235
Skip To Here 235
Stepping continuously 236

Working with the Data Window 236
Entering an expression 237
Editing expressions 238
Removing expressions 238
Setting values 238
Formatting values 238
Displaying and changing contexts 240
Evaluating expressions 240
Calling functions from the Data window 241
Working with expressions 241
Examining structs and arrays 242

Error Messages in the Debugger 243

Saving the Debugger State 246

Debugging Optimized Code 246
For both THINK C and Symantec C++ 246
For THINK C 247
For Symantec C++ 247

Debugging Options 247
Use Source Debugger 248
Use second screen. 248
Update program windows 248
Always save session 248
Allow dataview function calls 248
Store debug info separately. 249

Using Low-Level Debuggers 249
Using the Monitor command with TMON Pro 249
Using the Monitor command with other debuggers. . . . 250
Leaving the low-level debugger 250

Quitting the Debugger 250

Memory Considerations 250

Running with the Debugger

The debugger runs as a subordinate application with the THINK Project Manager. Although there is a debugger document icon, you can't launch it from the Finder by itself. When you set the option to use the source debugger, the THINK Project Manager takes care of launching it. Make sure that the THINK Debugger file is in the same folder as the THINK Project Manager.

*Another way to turn the debugger on is to check the Use Debugger check box in the Debugging section of the **THINK Project Manager options** dialog.*

Turning the debugger on

To run your application with the debugger, choose the **Use Debugger** command in the **Project** menu. This command turns the Use Debugger option on and off. When the option is on, the THINK Project Manager adds a "bug" column to the project window.

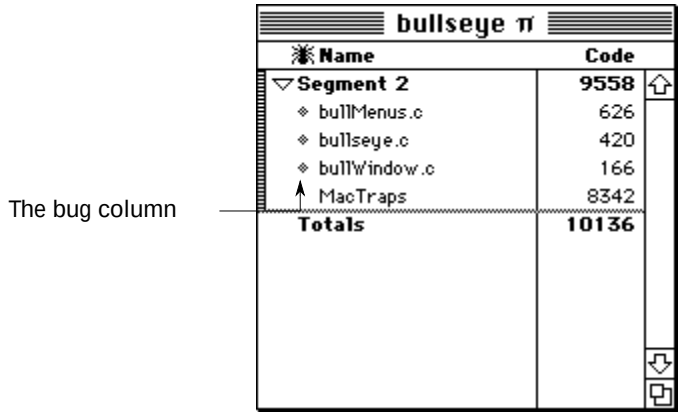

Figure 12-1 A project window with a bug column

The THINK compilers generate debugging information for source files that have gray diamonds next to them. Initially, all source files that contribute code get gray diamonds. The THINK compilers never generate debugging information for libraries or projects used as libraries.

Note

Not all THINK compilers generate debugging information. For example, THINK Rez and the `.o` Converter do not support debugging.

Clicking the bug column next to a file name turns the gray diamonds on and off. If you hold down the Option key as you click a gray diamond, the THINK Project Manager removes the gray diamonds from every file. If you hold down the Option key as you click the bug column where there isn't a gray diamond, the THINK Project Manager turns the diamonds on for every file that supports debugging in the project.

Running the project

When you run your program, the THINK Project Manager launches the source debugger instead of launching your program. The debugger then gets ready to run your program.

The debugger displays its own menu bar and two windows at the bottom of your screen. If you're using a Macintosh with more than one screen, and you have the Use Second Screen option on, the two debugger windows appear in the second screen.

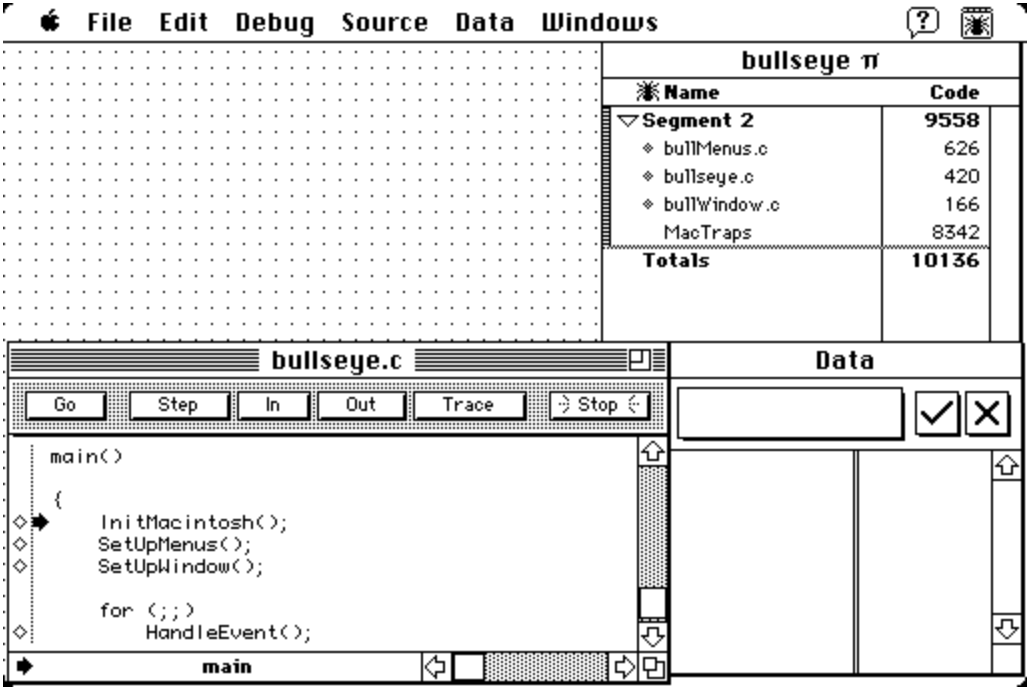

Figure 12-2 Running a program with the THINK Debugger

The larger window on the left is the Source window. It contains the debugger status panel and the source text of your program. The window on the right is the Data window. Use the Data window to display and change the values of your variables.

When your application is running, the screen shows the application's menu bar, and its windows are brought forward. When your application stops (at a breakpoint, for example), the debugger's menu bar appears, and its windows are brought forward.

The Debugger Windows

The debugger windows show you the source of your program and the values of your variables. Since the debugger works like any other application running under System 7 or MultiFinder, you need to be aware of whether it is your application or the source debugger in the foreground. Just because the debugger windows are frontmost doesn't mean that your program isn't running. In fact, if your application can run in the background under System 7 or MultiFinder, it will continue doing its background processing even if the debugger is the active application.

The Source window

The Source window contains the source text of your program, the debugger's status panel, statement markers, the current statement arrow, and the current function indicator. The title of the Source window is the name of the source file.

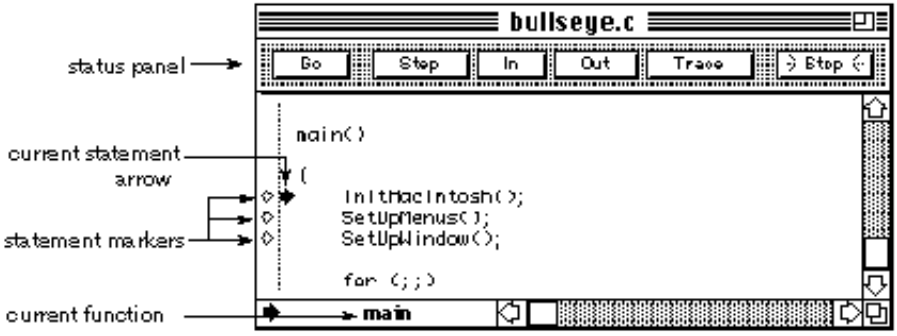

Figure 12-3 The Source window of the THINK Debugger

See "Working with the Source Window" later in this chapter.

The Source window shows the source text of your program. When you start the debugger, this window shows the file that contains the `main()` routine of your application.

See "Controlling Execution" later in this chapter.

The top of the Source window has a six-button status panel. These buttons control the execution of your program. The names of these buttons match the commands in the debugger's **Debug** menu.

See "Setting Breakpoints" later in this chapter.

The column of diamonds running along the left side of the source text are statement markers. Every line of your program that generates code gets a statement marker. You can set breakpoints only at statement markers.

The black arrow to the right of the statement markers is the current statement arrow. This indicator shows you the current statement, the one that the debugger is about to execute. When you first start your program, the current statement arrow is at the first executable line of your program.

See "The current function indicator" later in this chapter.

The source debugger uses the space at the lower left of the source window for the name of the current function. When you click here and hold the mouse button down, the debugger displays a pop-up menu that shows the call chain—the names of the functions that were called to get to the current function.

The Data window

The other debugger window is the Data window. In this window, you can examine and set the values of your variables.

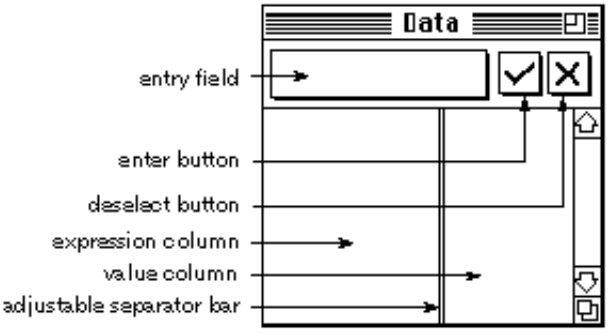

Figure 12-4 The Data window of the THINK Debugger

The Data window is modeled after a spreadsheet. Expressions you type into the entry field appear in the left column when you click the

enter button (check mark) or when you press the Return or Enter key. Pressing the Enter key leaves the expression selected. Pressing the Return key leaves the entry field empty so you can type in the next expression. The Enter button works just like the Enter key.

If you change your mind and don't want to enter an expression, click the Deselect button (X mark). The expressions you enter appear in the left column and their values are displayed in the right column.

You can select items in either the expression column or in the value column. If it's legal to edit the item, you can use the entry field to edit the item.

You can drag the center bar to make a column wider.

To clear an expression in the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key. To clear all the expressions, choose **Clear All Expressions**.

Working with the Source Window

The debugger gets the text for the Source window directly from your source file, so you see the file exactly the way you wrote it. As you step through your code and move from file to file, the debugger gets the text for the files it needs. The Source window only displays source text. You can select text to copy, but you can't change it.

The debugger displays the source text only if the compiler generated debugging information for it. If you step into a function that's in a file that didn't have a gray diamond next to it in the project window, the debugger displays the message `Debugging information not available`.

See "Editing files while debugging" later in this chapter.

If a source file is missing, or if you've edited a file while you're debugging, the debugger displays the message `Source text not available`.

The current statement and the selected statement

The current statement is the one that the debugger is about to execute. The current statement arrow always points to the current statement.

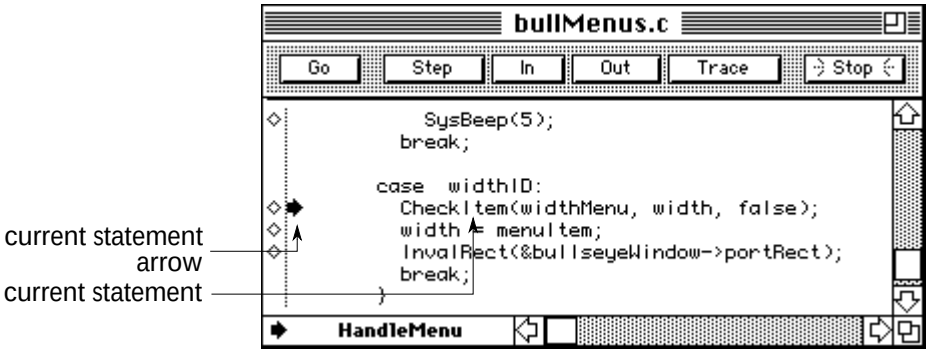

Figure 12-5 The current statement

The current statement arrow is hollow when there are still some instructions left to execute in a statement. (Remember that a single line in your source code may be compiled into several assembly instructions.) You may see hollow arrows when the statement is making an assignment, in a `for` statement, or cleaning up the stack after stepping out of a function.

Some of the debugger commands require you to select a statement in the Source window. To select a statement, just click once anywhere on its line. This line is called the selected statement. Note that you can select only lines with statement markers.

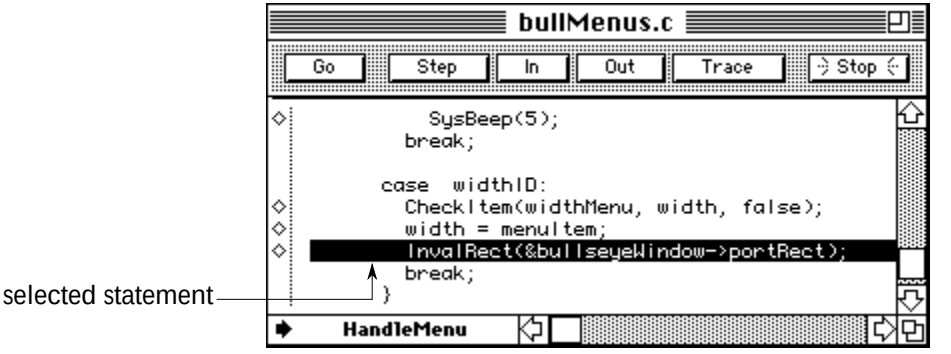

Figure 12-6 The selected statement

All commands that work on the selected statement work on the current statement if there isn't a selected statement.

If the current statement arrow isn't visible (because you've scrolled away or because you're viewing another source file in the Source window), click on the current function indicator in the lower left of the source window, or press the Enter key. The source file that contains the current statement will appear in the Source window.

The current function indicator

The current function indicator at the lower left of the Source window displays the name of the function in which the current statement is found. If the current statement is in a library, the current function indicator displays the name of the file. If the current statement is in ROM or outside your application's heap, the current file indicator displays the address of the program counter.

When you click and hold the mouse button down on the current statement indicator, you'll see a pop-up menu that shows you the call chain. The call chain follows stack frames through register A6, so if a function doesn't generate a stack frame, you won't see it in the call chain. In the Debugging pages of the `THINK C options` and `Symantec C++ options` dialogs, there is the Always generate stack frames option, which generates stack frames for almost all functions.

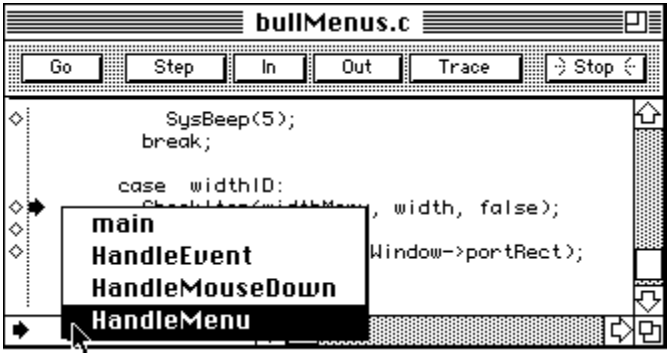

Figure 12-7 The call chain pop-up menu

If you choose an item in the call chain pop-up menu, the debugger opens the file that contains the selected function and selects the line to which the called function would return.

Viewing other source files in the Source window

The Source window usually shows the file that contains the current statement. To look at another source file in the Source window (to set breakpoints in it, for example) you tell the THINK Project Manager to send the text to the debugger:

1. Click the THINK Project Manager Project window.
2. Click the name of the file you want to look at.
3. Choose **Debug** from the THINK Project Manager's **Source** menu.
4. The file that you chose appears in the Source window.

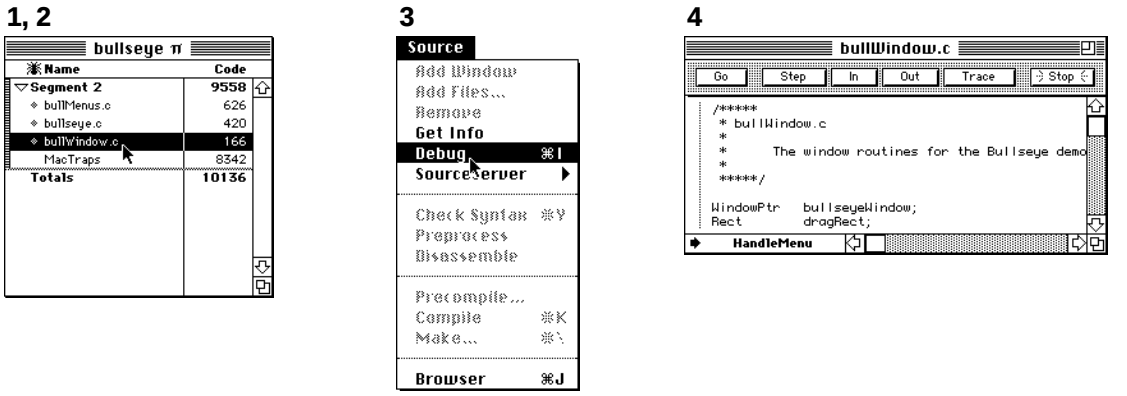

Figure 12-8 Viewing another file in the Source window

Now you can examine the file and set breakpoints in it. To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Note

If you select text in the source file before you choose **Debug** from the **Source** menu, the THINK Debugger scrolls the file so the selected text is visible in the Source window.

Viewing header files in the Source window

If a Symantec C++ header file contains executable code (a class definition with inline functions, for example), you can bring it into the Source window to set breakpoints or step through statements.

Note

You can view a header file in the debugger only if it's included in a Symantec C++ source file. THINK C does not store debugging information for header files.

To view a header file, first make sure that the file in the Source window includes it. Then hold down the Option key and click the title bar of the Source window. You'll see a menu, like the one in Figure 12-9, that lists the source file you're looking at and the header files that contain executable code, like `intstack.h`. It doesn't contain header files that contain only declarations, like `stdio.h`. Choose the header file you want to view. Notice that all executable statements in the header file, including inline functions, are marked with a diamond.

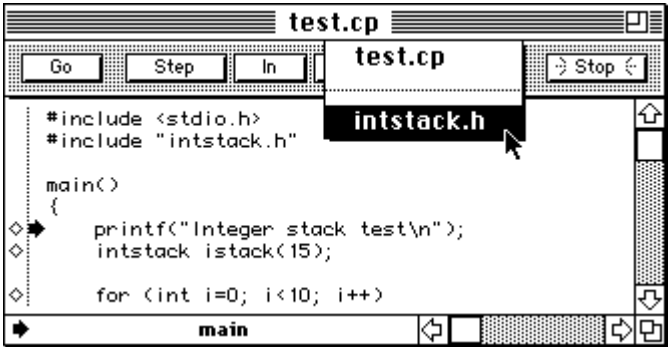

Figure 12-9 Going to a header file

Note

If a precompiled header contains an inline function, the header appears in the pop-up menu. If you select it, the Source window does not display the precompiled header. It continues to display the source file.

When you want to go back to the source file or go to another header file, hold down the Option key and click the title bar of the Source

window. You'll see the same menu as before that lists the source file and its header files. Choose the file you want to view.

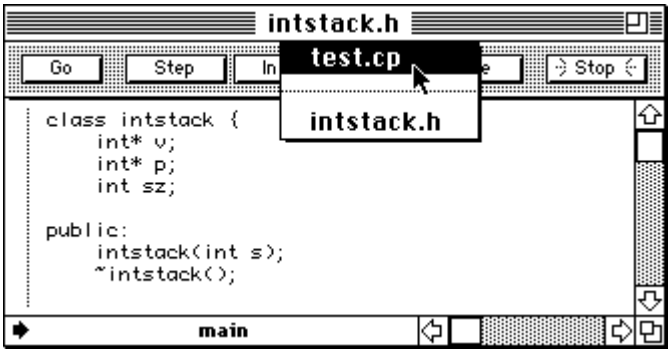

Figure 12-10 Going back to the source file

If a header file is included in more than one source file, each source file has its own copy of the header file's debugging information. So when you set a breakpoint in a header file, it's stored only in the source file whose name you see when you Option-click the Source window.

For example, suppose the source files `test.cp` and `util.cp` both include the header file `intstack.h`. If you're viewing `test.cp` in the Source window and go to `intstack.h` to set a breakpoint, that breakpoint is stored in `test.cp`'s copy of the debugging information but not in `util.cp`'s. Functions in `test.cp` that call `intstack.h` functions will stop at the breakpoint, but functions in `util.cp` that call `intstack.h` functions will *not* stop at the breakpoint.

Editing files while debugging

Because the THINK Project Manager is still running, you can edit your source files while you're debugging. To edit the file that's in the Source window, choose the **Edit '*filename.c*'** command from the **Source** menu.

To edit any other file in your project, click on the project window (or choose your project name from the **Windows** menu), and edit your files normally.

Naturally, the changes you make to source files won't take effect until you recompile.

Whenever the debugger needs the source text for a file, it looks for it on disk. The debugger caches the contents of the file as long as it can, but if it needs the memory for something else, it may reclaim the cache space. If you edit and save a file that the debugger has displayed in the Source window, it will continue to display the unedited version of the file as long as it's still cached. If the debugger sees that the source file on disk and the object code in the project don't match, it displays the message `Source text not available` in the Source window.

Searching in the Source window

To search for something in the Source window, you use the THINK editor.

1. Choose **Edit '*filename.c*'** from the **Source** menu (or press Command-E) to open an Editor window for the file in the Source window.

2. Use the THINK Project Manager **Find** command to find what you're looking for.

3. Choose **Debug** from the THINK Project Manager **Source** menu (or press Command-I) to get back the source debugger.

4. The line containing the selection in the editor window is displayed in the Source window.

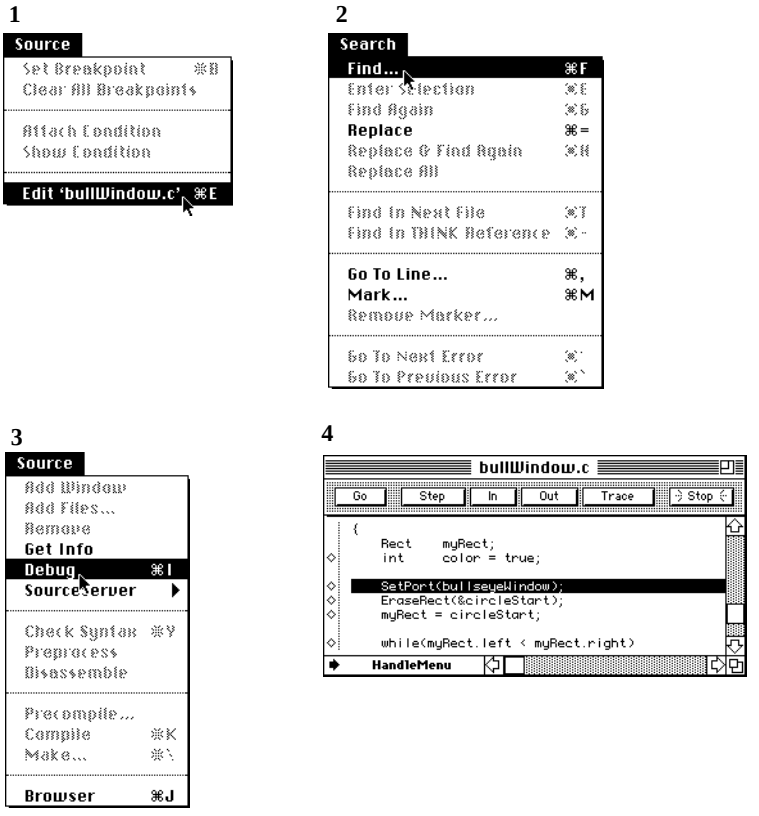

Figure 12-11 Searching from the Source window

*See "Controlling Execution" later in this chapter to learn about the **Go Until Here** command.*

When the Source window displays what you're looking for, you'll usually want to use the **Go Until Here** command.

Setting Breakpoints

The THINK Debugger lets you set breakpoints at any diamond statement marker. When your program is running, the debugger stops execution just before a breakpoint.

You can set three kinds of breakpoints: simple breakpoints, conditional breakpoints, and temporary breakpoints. Execution always stops at a simple breakpoint. Conditional breakpoints let you use the value of an expression in the Data window to determine

whether execution should stop. Temporary breakpoints let you set a breakpoint and start execution in a single step. When your program reaches a temporary breakpoint, execution stops, and the debugger automatically clears the temporary breakpoint.

You can set breakpoints while your program is running, not just when it's stopped.

You can also use the functions `Debugger()` and `DebugStr()` to force your program to stop and drop into the debugger.

Simple breakpoints

To set a breakpoint, click a statement marker diamond. The diamond will change from hollow to filled, indicating that the breakpoint is set.

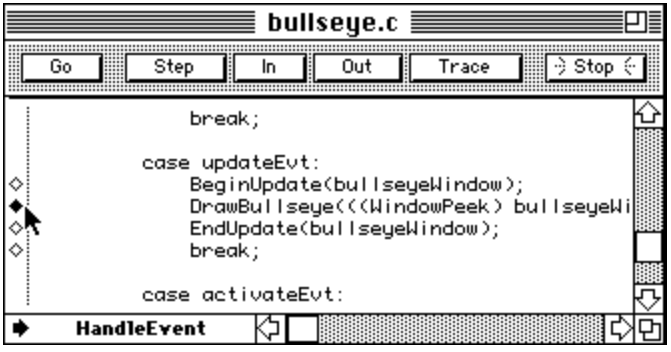

Figure 12-12 Setting a breakpoint

To clear a breakpoint, click the filled diamond. The statement marker changes to a hollow diamond to show you that the breakpoint is clear. The **Clear All Breakpoints** command in the **Source** menu clears every breakpoint in every file in the project.

You can also use the **Set Breakpoint** and **Clear Breakpoint** commands in the **Source** menu to set and clear breakpoints.

1. Click a statement in the Source window to select it.
2. Choose **Set Breakpoint** or **Clear Breakpoint** from the **Source** menu to set or clear a breakpoint.

Setting breakpoints in another file

The Source window usually shows the file that contains the current statement. To set breakpoints in another file, you must first tell the THINK Project Manager to send the file to the debugger:

1. Click the THINK Project Manager's Project window. (Or choose your project name from the **Windows** menu.)

2. Click the name of the file in which you want to set the breakpoint.

3. Select **Debug** (Command-I) from the THINK Project Manager's **Source** menu. The source debugger's Source window will display the text of the file you chose.

4. Click statement marker diamonds to set breakpoints.

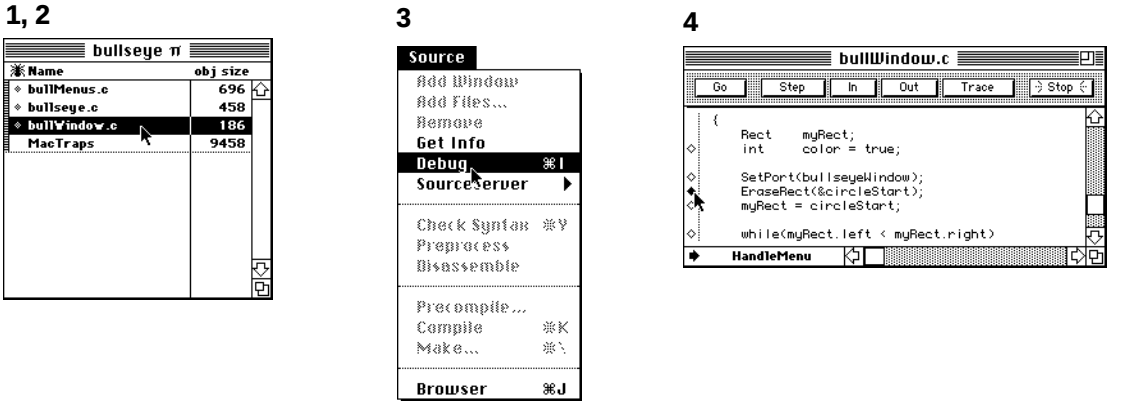

Figure 12-13 Setting a breakpoint in another file

To get back to the current file, click the current function indicator at the lower left of the Source window or press the Enter key.

Setting temporary breakpoints

Temporary breakpoints are useful if you want to go quickly to a particular line of your program. For instance, suppose you wanted to examine how your program handles menu selections. You'd set a temporary breakpoint at the first line of your menu-handling routine. The program would run normally until you chose a command from one of your menus.

To set a temporary breakpoint, hold down the Command or Option key as you click a statement marker. When you release the mouse button, the debugger starts your program, and execution continues until you hit any breakpoint, not just the temporary breakpoint. The source debugger clears all the temporary breakpoints when you stop for any reason.

Note

The **Go Until Here** command sets a temporary breakpoint at the selected line.

Setting conditional breakpoints

The THINK Debugger lets you set conditional breakpoints. A conditional breakpoint is a breakpoint that has a condition associated with it. The debugger stops execution at conditional breakpoints only when the condition evaluates to non-zero.

To set a conditional breakpoint:

1. Set a breakpoint by clicking the statement marker diamond.
2. Click the line to select it.
3. Click an expression in the expression column of the Data window.
4. Choose **Attach Condition** from the **Source** menu.

5. The breakpoint marker turns into a gray diamond to indicate that it's a conditional breakpoint.

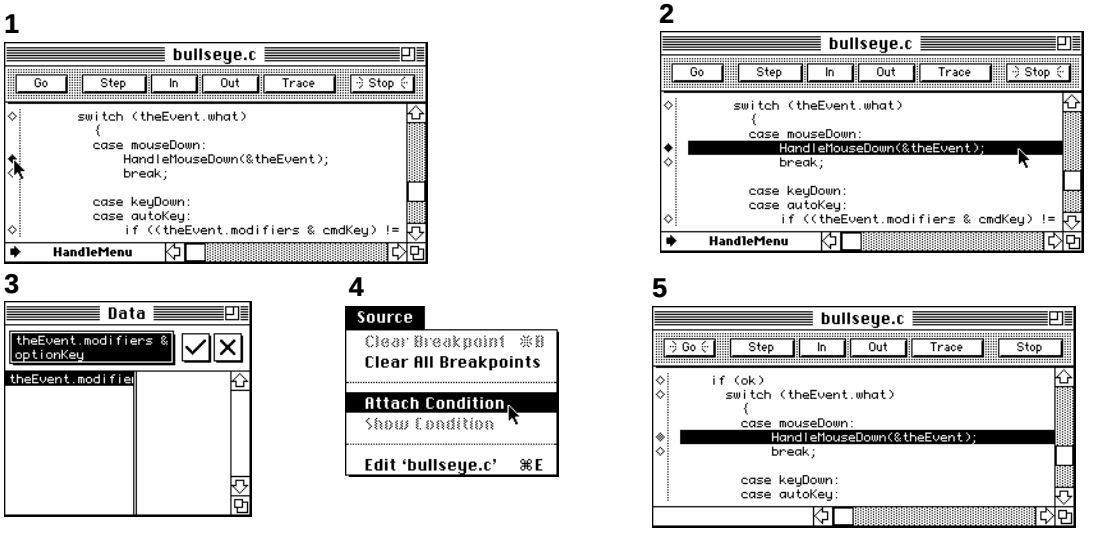

Figure 12-14 Setting a conditional breakpoint

In this example, the debugger will stop at the breakpoint only if you hold down the Option key as you click (`theEvent.modifiers & optionKey == 0x0800`).

If you're already stopped at a simple breakpoint that you want to turn into a conditional breakpoint, just select an expression in the Data window, and choose **Attach Condition**. Since the debugger uses the current statement if there isn't a selected line, the condition will be attached to the current statement.

The **Attach Condition** command is dimmed if:

- The expression can't be evaluated in the context of the breakpoint
- A breakpoint isn't set
- An expression in the Data window isn't selected
- The expression is a floating-point expression

As you debug your program, you may forget the condition that's associated with the breakpoint. To see the condition associated with a conditional breakpoint:

1. Click on the line to select it.
2. Choose **Show Condition** from the **Source** menu.

The condition associated with the conditional breakpoint will be selected in the Data window.

To learn how to use the Data window, see "Working with the Data Window" later in this chapter.

Using Debugger() and DebugStr()

The functions `Debugger()` and `DebugStr()` cause your program to stop and drop into the debugger just as if you had set a breakpoint. The `DebugStr()` function takes a Pascal string as an argument. The string appears in place of the status panel. For an example of `DebugStr()`, see "Error Messages in the Debugger" later in this chapter.

If you're not using the THINK Debugger, these functions cause your program to drop into a low-level debugger if one is installed. If no debugger is installed, your Macintosh may crash with an "unimplemented trap" error.

Controlling Execution

The debugger uses six commands to control execution. To make it easier to debug your programs, there are three ways to use them: you can choose them from the **Debug** menu, you can use the Command key equivalents, or you can use the buttons in the status panel of the Source window.

The buttons in the status panel do double duty as status indicators. When your program is running, the Go button is lit. When your program is stopped, the Stop button is lit. Remember that your program can still be running even if the debugger windows are frontmost.

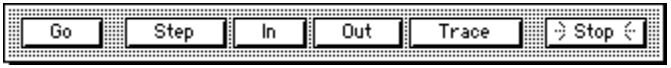

Figure 12-15 The status panel

Go

The **Go** command starts your program if it was stopped. Your program runs until you stop it (with the Stop button, for example), until it's about to execute a line with a breakpoint, or until it hits an exception (dividing by zero, for instance). If your application is already running, the Go command brings it to the foreground.

Trace

The **Trace** command executes the current statement. In most cases, the statement indicator will go on to the next statement marker, even if the next statement marker is in another function. The only time it won't is when the program counter steps into some code for which the debugger doesn't have the source text. This usually happens when you step into a trap that's not generated inline. So, for a brief period, the current statement arrow isn't really anywhere in your program, but somewhere in `MacTraps` instead. Though you can't see the current statement arrow, the current function indicator at the lower left tells you which file it's in.

Step

The **Step** command is like the **Trace** command except that it goes on to the next statement marker in the current function. If you're at the end of a function, **Step** returns to the calling function. Use **Step** when you want to follow the execution within a function without falling into another function. (Technically speaking, Step skips over `JSR`s.)

The first time you step through a `for` loop, the current statement arrow jumps to the end of the loop because in the generated code, that's where the test is done.

Step In

The **Step In** command executes Trace commands until the current statement arrow falls into a function. **Step In** is useful when you want to skip over a set of assignments or Toolbox traps, to fall into the next function call. If **Step In** reaches the last statement of the current function without falling into another function, it will stop immediately after the function returns.

Step Out

The **Step Out** command executes **Step** commands until the current statement arrow falls out of the current routine. This operation can be slow if there's a lot left to do, but it's a sure way of leaving the current routine. A faster way of leaving the current routine is to use

the **Go Until Here** command or to set a temporary breakpoint at the last diamond in the function.

Stop

The **Stop** command stops execution of your program. The **Stop** command works when any debugger window is active, and the Stop button only works when the Source window is the active window. When you click the Stop button you'll usually be coming out of your call to `GetNextEvent()` or `WaitNextEvent()`.

If your program is not in its event loop, you might not be able to make the debugger the frontmost application. In this case, Command-Shift-Period is the panic button. Use Command-Shift-Period to stop execution when one of your application's windows is frontmost or when you think it's stuck in a loop.

Warning

Command-Shift-Period won't work if you're stuck in an infinite loop in ROM.

Go Until Here

The **Go Until Here** command starts execution and stops at the selected line. This command works exactly the same as setting a temporary breakpoint (see "Setting Breakpoints" earlier in the chapter) at the selected line. Use this command when you want to move through a block of code quickly.

See "Searching in the Source window" earlier in the chapter to learn how to look for text from the debugger.

Using this command is more convenient than setting a temporary breakpoint when the line you want to go to is already selected. For instance, it's easier to press Command-H after you've found a string you're looking for.

Skip To Here

The **Skip To Here** command changes the program counter to the selected line without executing any intervening code. Use it when you want to skip over code you know to be buggy but not crucial to the rest of the program's operation.

Warning

This command is potentially dangerous. Make sure the code you're skipping to doesn't depend on anything the skipped code does. For instance, it is a very bad idea to skip over initialization routines.

Stepping continuously

If you hold down the Option or Command key as you click one of the status panel buttons (except Stop), you'll enter auto mode. In auto mode, the debugger updates the Source and Data windows and repeats the command. To trace through every line of your program automatically, for example, hold down the Option key as you click the Trace button.

One useful technique is to set breakpoints at spots where you'd like to look at some variables and then do an Auto-Go. When your program hits the breakpoint, the debugger will update the Data window and start the program up again.

To cancel auto mode, type Command-Shift-Period. The debugger will finish the command and then stop. Note that if you're in Auto-Go mode, the Stop button just cancels auto mode. You'll have to click the Stop button again to stop your program.

Working with the Data Window

The Data window lets you examine and modify the values of your variables as you debug your programs. You can type any legal expression for the current file's translator (C or C++) in the entry field. The expression you type appears in the expression column, and its value appears in the value column. You can display values in

several formats to make your debugging easier. You can also display structures and arrays in their own windows.

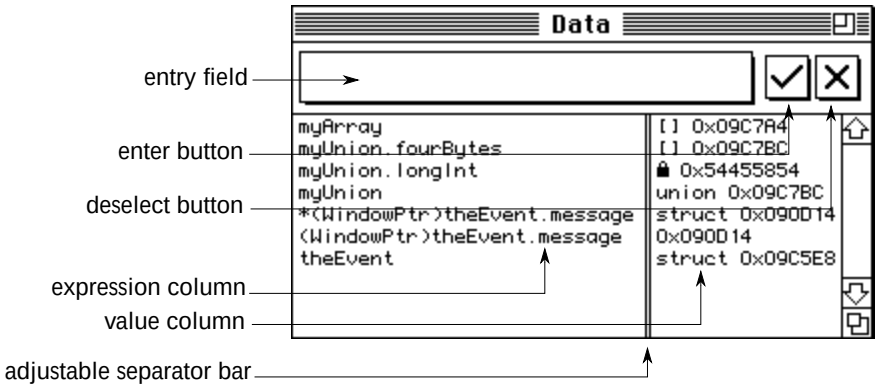

Figure 12-16 The Data window

Note

The first time you enter a C++ expression, there may be a noticeable delay before the Data window displays its value. The debugger will evaluate subsequent expressions more quickly.

Entering an expression

For more information on the Allow dataview function calls option, see "Calling functions from the Data window" later in the chapter.

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements or expressions involving ++, --, +=, and so on. You can't evaluate function calls unless you turn on the Allow dataview function calls option.

The debugger compiles the expressions you enter in the context of the selected line in the Source window, or, if you haven't selected a line, in the context of the current statement. If the expression you enter is not defined within the context, the debugger displays an error message as the value of the expression.

The quickest way to enter an expression into the Data window is to select it in the Source window and choose the **Copy to Data** command in the debugger's **Edit** menu.

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with

dynamic storage—in other words, if it refers to non-static variables local to a function. All other expressions have global scope.

If you use the Enter key to enter an expression, the expression remains selected. If you use the Return key to enter an expression, the entry field is ready to accept the next expression. Clicking the Enter button is the same as pressing the Enter key.

Editing expressions

To edit an expression in the Data window, select it in the left column. The expression will appear in the entry. Use standard Macintosh editing techniques to edit the text of the expression.

When you edit an expression in the Data window, the context is the same as when you entered the expression. To change the context of an expression to the current context after you edit it, hold down the Option or Command key as you click the Enter button.

Removing expressions

To remove an expression from the Data window, select the expression you want to remove and choose the **Clear** command from the **Edit** menu or press the Clear key. To remove all the expressions from the Data window, use the **Clear All Expressions** command in the **Data** menu.

Setting values

The debugger lets you change the values of your expressions as long as the expression would be legal in the left side of an assignment statement.

Formatting values

When you enter an expression, it's added to the left column. The value of the expression appears in the right column. You can use the formats in the **Data** menu to change how the debugger displays the variables.

To change a format, select an expression in the Data window. Then choose a format from the **Data** menu. Some of the formats won't be

available. The formats available depend on the type of the expression. The default formats are shown in italics.

Type	Formats Available
integers	*Signed Decimal*, Unsigned Decimal, Hexadecimal, Character
unsigned	*Unsigned Decimal*, Signed Decimal, Hexadecimal, Character
pointers	*Pointer*, Address, Hexadecimal, C String, Pascal String
arrays	*Address*, C String, Pascal String
structs	Address
unions	Address
functions	Address
floats	Floating Point

Table 12-1 Display formats available

This is what the display formats look like:

Format	Example
Signed Decimal	`4523345`, `-1`
Unsigned Decimal	`4523345`, `65535`
Hexadecimal	`0xA09E1487`
Character	`'c'`, `'TEXT'`
C String	`"abcdef\nghi\33"`
Pascal String	`"\pabcdef\nghi\33"`
Pointer	`0x7A7000` or `0x007A7000`
Address	`[] 0x09FE44`, `struct 0x08FC14`
Floating Point	`1961.0102`

Table 12-2 Display format examples

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise, it uses `\nn`, where `nn` is an octal value.

Of course, you can use typecasting to use formats that aren't normally available. For example, if you really want to see the integer `i` as a C string, you would type this expression: `(char *) i`.

To see any pointer as an array, just change its format to Address. This way, when you double-click its value, you'll see an array window instead of the value of what the pointer points to.

Displaying and changing contexts

If you forget the context of an expression in the Data window, use the **Show Context** command in the **Data** menu. The Source window will display the context for the expression.

When you select an expression, you can edit it in the entry field. When you click the Enter button (or press the Return or Enter key), the debugger will recompile the expression in its original context.

To change the context of an expression you've already entered or edited, select the context in the Source window, select the expression, then choose **Set Context** from the debugger's **Data** menu. You can do the same thing by holding down the Option or Command key as you click the Enter button (or press the Return or Enter key).

Sometimes you want an expression to be context-free. For example, you might use the same variable name in several different routines. When you enter a routine you'd like to see the value of the variable in the new routine instead of a blank space. To make an expression context-free, select it in the Data window and choose **Context Free** from the **Data** menu. The debugger marks it with a small arrow, the same arrow that points to the current statement in the Source window, to show that the variable's context is the same as the context of the current statement.

Evaluating expressions

The debugger re-evaluates the expressions in the Data window every time your program stops. Expressions with a context that isn't in the current function are not re-evaluated unless they have global scope. To keep the Data window from getting too cluttered, the debugger clears their values.

Sometimes you don't want an expression to be re-evaluated. For instance, you might want to compare the values of the same expressions at different times. Or you may want to keep the value of

an expression in the Data window after the expression goes out of scope. Select an expression and choose **Lock** from the **Data** menu. The debugger marks it with a lock icon.

Calling functions from the Data window

To call functions from the Data window, turn on the Allow dataview function calls option in the Debugging page of the **THINK Project Manager options** dialog.

Warning

Use this feature carefully. Calling functions carelessly could crash your program and cause the Macintosh to hang.

Avoid calling functions that have side effects. Good functions to call include `strcmp()` and any function that returns a value without modifying data or its arguments. You can call some Macintosh Toolbox routines, like `GetHandleSize()`, or `oops` library routines like `class_name()`, although they may change the value of `MemErr`, a low memory location.

Because the THINK Debugger does not expect side effects, the Data window may not display correct values if a function call has a side effect. For example, if you display `MemErr` and then `GetHandleSize(h)`, the THINK Debugger will display the value that `MemErr` had before you called `GetHandleSize()` and will not update it until you step again.

Also, the Data window can't display functions that are declared `void`.

Warning

Never call a function that doesn't return, such as `longjmp()` or `ExitToShell()`. Your machine will hang.

Working with expressions

Double-clicking the left column of the data window makes a copy of the expression. This is useful if you want to lock one copy down while you let another be evaluated every time the debugger stops.

Double-clicking the value of struct, union, or array opens up a new window.

Double-clicking a value formatted as Pointer enters a new, dereferenced expression in the Data window. If you hold down the Shift key, the new expression will be dereferenced twice.

If you hold down the Option key as you double-click, the new entry replaces the original entry.

Examining structs and arrays

You can examine fields of structs and arrays displayed in the Data window. (In this section, whatever you read about structs applies to classes and unions as well.)

When you double-click the right column on an expression whose value is struct, the debugger opens up a Struct window. The Struct window looks like the Data window. The names of the fields appear in the left column, and their values appear in the right column.

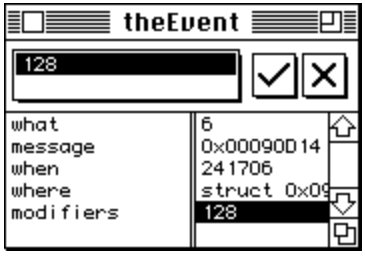

Figure 12-17 A Struct window

You can change the values of the fields of the records the same way you change any variable. Of course, you can't change the names of the fields.

Double-clicking a value opens other windows the same way as double-clicking the main data window. A new expression appears in the main Data window for the new window. Double-clicking a field name enters a new expression in the main Data window.

Note

All Struct and Array windows "belong" to the main Data window. For every Struct or Array window, there is an entry in the main Data window.

As mentioned above, Option-double-clicking replaces the original expression with the new one. For example, suppose you had a

Struct window for an `EventRecord`, `theEvent`. Option-double-clicking on the `what` field would make the Struct window disappear (the original expression is `theEvent`), and `theEvent.what` would appear in its place in the Data window.

Array windows are similar to Struct windows. The indices appear in the left column and values appear in the right column. Unlike the main Data window or Struct windows, every element in an array is displayed in the same format.

Because C and C++ compilers don't enforce array bounds, array windows have "infinite" scroll bars.

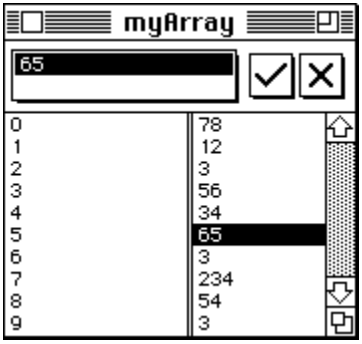

Figure 12-18 An Array window

If you select an index in the left column and change its value, the debugger will display the array from that index.

To see a pointer as an array, set its format to Address and double-click its value.

When you have several Struct or Array windows open, it's easy to lose track of them. To bring a particular Struct or Array window to the front, you can choose its name from the **Windows** menu.

You can close all the Array or Struct windows at once. Choose **Close All** from the **File** menu or hold down the Option key as you click the Close box of any Struct or Array window.

Error Messages in the Debugger

If executing a statement in the Source window or evaluating an expression in the Data window would cause a run-time error, the

THINK Debugger displays an error message in the status panel or in the value column of the Data window.

In Figure 12-19, the Source window contains a small program that tries to access illegal memory (a negative memory address, in this case). The Data window has an expression that tries to divide by zero and one that tries to dereference the illegal pointer.

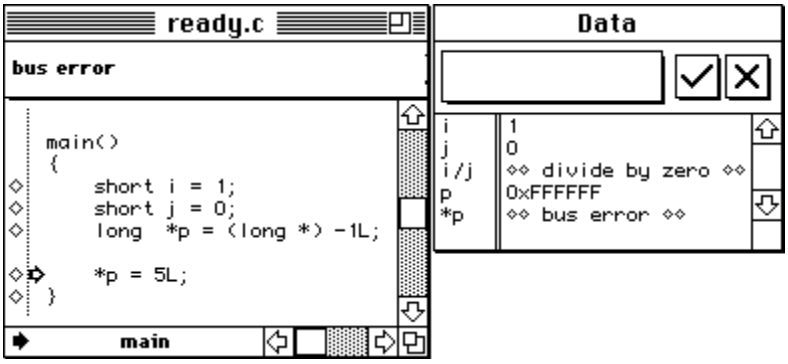

Figure 12-19 Errors reported in the debugger

Here are some of the errors you might see in the debugger:

Exception	Occurs when your program...
bus error	Writes to or reads from an address that doesn't exist.
odd address	Writes to or reads from an odd address on a 68000 machine, when the size of the data being read or written is larger than 1 byte in size.
illegal instruction	Executes an unknown instruction. This can happen if the program you're running is compiled with the options Generate MC68020 instructions or Generate MC68881 instructions and you don't have an MC68020 or MC68881 machine.
divide by zero	Divides by zero.

Table 12-3 Debugger errors

Exception	Occurs when your program...
CHK exception	Encounters a CHK instruction that fails. CHKs are typically used by Pascal compilers to perform range checking. If a CHK exception occurs when running a C or C++ program, the effect is usually the same as when you get an illegal instruction error.
TRAPcc exception	Encounters an arithmetic overflow TRAPcc or TRAPV instruction. This instruction is also typically used by Pascal compilers and shouldn't occur in a C or C++ program.
bomb!	Overwrites the debugger's partition. Also occurs when a system error occurs inside the debugger itself, or when the debugger was trying to handle an exception on the program's behalf. This error usually only occurs after your program has encountered other errors.

Table 12-3 Debugger errors *(Continued)*

In most cases, the safest thing to do when you get an error in the Source window is to choose the **ExitToShell** command from the **Debug** menu. If you want to continue from the error, click the error message in the Source window, and the status panel will reappear.

If you use the `DebugStr()` function in your program, your program stops and the string appears the same way an error message would:

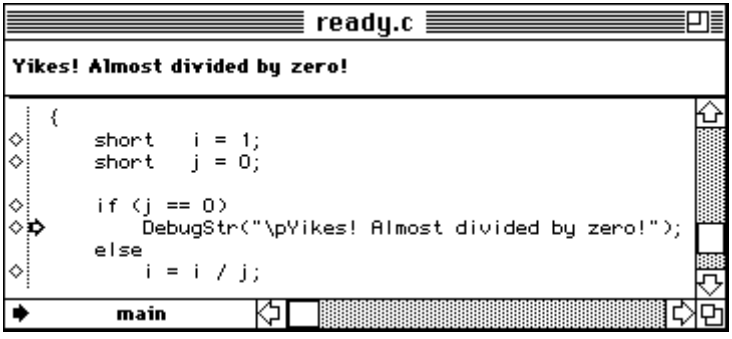

Figure 12-20 Using DebugStr() to report anomalous behavior

Saving the Debugger State

Use the **Save All** command in the debugger's **File** menu to save all of your breakpoints, the expressions in the Data window, and the positions of all your debugger windows. Whenever you start the debugger again, it restores the debugger state. If you don't want the debugger to restore the state for a particular session, hold down the Option key as the debugger starts up.

To prevent the debugger from saving your state, hold down the Option key as you quit the debugger.

To put the Debugger windows into their default positions, hold down the Shift key as you start the debugger.

By default, the debugger does an automatic save when the Always save session option is on in the Debugging page of the **THINK Project Manager options** dialog. If this option is off, you can still use the **Save All** command to save the debugger state. No matter how that option is set, the debugger always restores the previously saved state.

If you want to make sure that the THINK Debugger doesn't save any session information, use the **Clear All Breakpoints** command in the **Source** menu and the **Clear All Expressions** command in the **Data** menu. Then choose **Save All** from the **File** menu.

Editors that are THINK-compatible will update the debugger state information. See "Using a THINK-compatible editor" in Chapter 10.

The THINK compilers store some debugger state information directly in your source files. As you edit your files, the THINK editor maintains breakpoints and the contexts of Data window expressions. However, other editors may not update the debugger state information. If you edit files with debugger information in other editors, your breakpoints may be set at the wrong lines, and the Data window expressions may be evaluated in the wrong contexts.

Debugging Optimized Code

For more information about the THINK C optimizer, see Chapter 17, "Optimizing Code."

Code optimizers change the way that the THINK compilers compile your programs. You should be aware that optimized programs might behave differently under the debugger than they would if they were not optimized. This section tells you how some optimizations affect THINK C and Symantec C++ code.

For both THINK C and Symantec C++

If you use a global optimizer, certain statements might never get executed, so the compiler doesn't generate code for them. That means that you would not be able to set a breakpoint there. If you use the CSE Elimination (common subexpression elimination) option, changing the value of a variable in the data window might not make a difference if the subexpression it belongs to has already been evaluated.

For THINK C

If you have the Register Coloring option on, and the optimizer uses the same memory location for two separate variables, changing one variable in the Data window might change another one as well. If you use the Defer & Combine Stack Adjusts option in THINK C, you should not use the **Skip To Here** command in the debugger to skip over a function call. If you use the Suppress Redundant Loads option, changes that you make to values in the Data window might not "take" if the value was stored temporarily in a register.

For Symantec C++

If you use the Dead Variable Elimination option, you can't evaluate variables that were eliminated. Instead the Data window displays the message `variable was eliminated`.

Debugging Options

This section describes all the options that control the THINK Debugger. To set these options, go to the Debugging page of the **THINK Project Manager options** dialog.

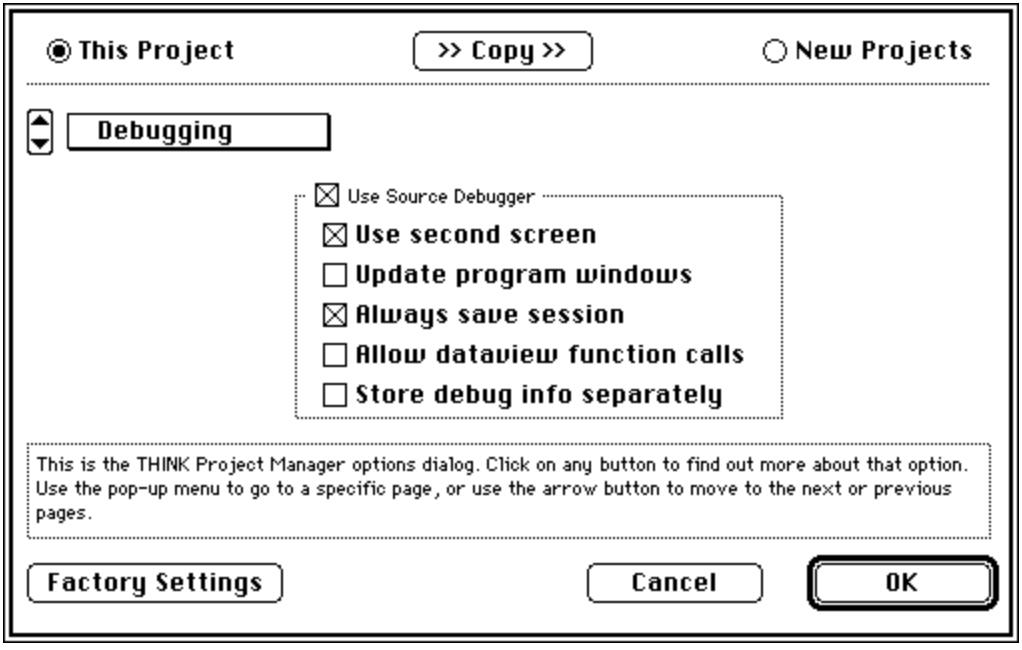

Figure 12-21 The debugging options

Use Source Debugger

This check box is the master switch for the THINK Debugger. If it's on, the THINK Project Manager launches the source-level debugger when you choose the **Run** command. Clicking this check box has exactly the same effect as choosing **Use Debugger** from the **Project** menu.

Use second screen

If you have more than one monitor, the debugger uses the second one to display the Source and Data windows. This option is available only when the Use Source Debugger option is on.

Update program windows

When this option is on, the debugger tries to update your windows for you when your project is stopped. If this option is off, your program must wait until control returns to it, so it can handle the update event itself. Since it can't do so until it gets back to its event loop, an update may remain pending for some time.

This option is useful if you're trying to step through code that draws in a window and the window is partially obscured by a THINK Project Manager or THINK Debugger window.

Because this window uses memory to save the window image, you might need to increase the THINK Debugger's memory partition if you're working on a large screen or in color. See "Memory Considerations" later in this chapter for more information.

Always save session

If this option is on, the THINK Debugger saves the state of your debugging session before the debugger quits. The effect is the same as if you had chosen the **Save All** command from the THINK Debugger's **File** menu. If this option is off, the debugger does not save the debugger state automatically at the end of each session. If you do a manual save, though, the debugger will restore that state when you start the debugger again. See "Saving the Debugger State" earlier in this chapter for more information about saving sessions.

Allow dataview function calls

If this option is on, you can call functions from the Data window. For more information, see "Calling functions from the Data window" earlier in this chapter.

Warning

Use this feature carefully. Calling functions carelessly could cause your program to crash and corrupt your hard drive!

Store debug info separately

For more information on shielded folders, see "Using shielded folders and project-specific folders" in Chapter 9.

The THINK Project Manager is noticeably slower when your project is over 5MB, and it can't handle projects over 16MB at all. A large portion of what's in your project is debugging information. To store the debugging information in separate files, choose the Store debug info separately option. The next time you run or build your project, the THINK Project Manager creates a shielded folder in your project tree and creates a debug file for each source file it compiles. The folder is named after your project with `XSYMs` appended. The debug files are named after the source files. For example, if the project `MyProject.π` contains `MyFile.c` and `MyTools.c`, the folder is named `(MyProject.π.XSYMs)` and the debug files are named `MyFile.c` and `MyTools.c`.

Using Low-Level Debuggers

The THINK source-level debugger is great for figuring out what your application is doing. But sometimes you need to get closer to the machine. The **Monitor** command in the **Debug** menu invokes a low-level debugger like TMON or Macsbug.

When you use the **Monitor** command, all registers and low-memory globals contain the proper values for your program. The PC (program counter), however, will not be pointing to the next instruction. Instead, it will be pointing somewhere in the debugger.

Warning

If you don't have a low-level debugger installed, don't use the **Monitor** command.

Using the Monitor command with TMON Pro

To display a window showing a disassembly of a program's code in TMON Pro, enter this line in a command window:

To type ∆, press Option-J.

```
new assembly ∆(PC-4)^
```

Using the Monitor command with other debuggers

If you're using another low-level debugger, like MacsBug, the correct value of the PC is right before the current PC. For instance, to display your program's code in MacsBug, type:

```
IL (PC-4)^
```

Leaving the low-level debugger

If you entered the low-level debugger through the **Monitor** command, use your debugger's standard exit command.

If you got into your low-level debugger any other way, there is no simple way to return to the source debugger. You can use your low-level debugger's **ExitToShell** to abort your program as long as it's the foreground application. Check the low-memory global `CurApName` (`0x0910`). If it contains the name of your program, then you can use the **ExitToShell** command.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

Memory Considerations

The THINK Project Manager, the source debugger, and your project each run in their own MultiFinder partition. The default partition sizes are:

Application	Partition Size
THINK Project Manager	2048K for THINK C 4000K for Symantec C++
THINK Debugger project	316K 400K

Table 12-4 Default partition sizes

If you want to run other applications, or if you find you're running out of memory, you can change the partition sizes. Change your project partition first. If you still don't have enough memory, change the size of the THINK Project Manager partition. Finally, change the debugger's partition.

The default partition size for your project is 384K. Moderately sized projects don't usually need this much memory. You might want to try smaller values so you can give more memory to other applications.

The THINK Project Manager uses the most memory when it's compiling. If the THINK Project Manager complains that it's running out of memory when you compile, close any open windows before rebuilding your project. You should also exit all other open applications to reduce disk cache size, or expand the partition size for the THINK Project Manager. The THINK Project Manager partition should be no smaller than 2048K for THINK C or 2500K for Symantec C++.

If you have the Update program windows debugger option on, the debugger partition might need to be bigger than 250K. You will definitely need to increase it if you're running with either large or color monitors. If this option is off and you're running a small project, you can make the debugger partition as small as 200K. To change the size of your project's partition, use the **Set Project Type** command from the **Project** menu. You can change the partition sizes for the THINK Project Manager and the debugger from their **Get Info** boxes in the Finder. You can't change the partition size of an active application, so you'll have to quit the THINK Project Manager first to change its partition size.